COS 333: Advanced Programming Techniques

• **today**
  – administrative stuff
  – course overview
  – regular expressions and grep

• **check out the course web page (CS, not Blackboard) and Piazza**
  – notes, readings and assignments are posted only on the web page
    monitor the web page and Piazza every day
  – Assignment 1 is posted; due midnight Friday Feb 17
  – initial project information is posted (much more on Thursday)

• **please do the survey if you haven't already**
People

Christopher Moretti
Sotiros Apostolakis
Jon Balkind
Nick Giannarakis
Qizhe Cai
Ghassen Jerfel
Robin Qiu
Gautam Sharma
Very Tentative Outline

week 1  regular expressions, grep; project info
week 2  scripting: shell, AWK, Python
week 3* web technology: HTTP, CSS, Javascript
week 4  client and server frameworks
week 5  phone apps
week 6  databases; software engineering
        (spring break)
week 7  user interfaces
week 8  networks
week 9  advanced C++, Java; Go
week 10* APIs; XML, JSON, REST
week 11 Design patterns. DSLs
week 12 ??
        (start of reading period)
May  8-12 project presentations in this week
May  14 project submission

* => guest
House rules

- please turn cell phones off
- please don't use your laptop, tablet, phone, ...
  - it distracts you
  - it distracts your neighbors
  - it distracts me
- please don't snore (sleeping is ok)
- please sit towards the front, not in the back
- please stay away if you're sick !!!

- please ask questions about anything at any time
Regular expressions and grep

• **regular expressions**
  – based on ideas from automata theory pioneered by Stephen Kleene *
  – notation
  – mechanization
  – pervasive in Unix tools
  – in all scripting languages, often as part of the syntax
  – in general-purpose languages, as libraries
  – basic implementation is remarkably simple
  – efficient implementation requires good theory and good practice

• **grep is the prototypical tool**
  – written by Ken Thompson @ Bell Labs ~1972
Grep regular expressions

c any character matches itself, except for
   metacharacters . [ ] ^ $ * \n
r_1r_2 matches r_1 followed by r_2
.
matches any single character
[ . . . ] matches one of the characters in set ...
   shorthand like a-z or 0-9 includes any character in the range
[^ . . . ] matches one of the characters not in set
   [^0-9] matches non-digit
^ matches beginning of line when ^ begins pattern
   no special meaning elsewhere in pattern
$
matches end of line when $ ends pattern
   no special meaning elsewhere in pattern
*
any regular expression followed by * matches 0 or more
\c matches c unless c is ( ) or digit
\( . . . \) tagged regular expression that matches ...
   the matched strings are available as \1, \2, etc.
Examples of matching

- `xy`  
  - `xy` anywhere in string

- `^xy`  
  - `xy` at beginning of string

- `xy$`  
  - `xy` at end of string

- `^xy$`  
  - string that contains only `xy`

- `^`  
  - matches any string, even empty

- `^$`  
  - empty string

- `.`  
  - non-empty string, i.e., at least 1 char

- `xy.$`  
  - `xy` plus any char at end of string

- `xy\.$`  
  - `xy` at end of string

- `\\xy\`  
  - \xy\ anywhere in string

- `[xX]y`  
  - `xy` or `Xy` anywhere in string

- `xy[0-9]`  
  - `xy` followed by one digit

- `xy[^0-9]`  
  - `xy` followed by a non-digit

- `xy[0-9][^0-9]`  
  - `xy` followed by digit, then non-digit

- `xy1.*xy2`  
  - `xy1` then any text then `xy2`

- `^xy1.*xy2$`  
  - `xy1` at beginning and `xy2` at end
Back-referencing:

using what was matched before \(...\) \(...\) ... \1 \2 ... \9

grep '\(ious\)\).*\1' web2
  homoiousious

grep '\(\.\)\).*\1.*\1.*\1.*\1.*\1.*\1.*\1' web2
  electrotelethermometer
  possessionlessness
  stresslessness
  successlessness

grep '\(i\(\.\)\2\).*\1' web2
  flibbertigibbet
  hillbilly
  killeekillee
  kinnikinnick
  maxillopremaxillary
  Mississippi
  Mississippian
  tirralirra
  tirrwirr

back-referencing is not "regular"; not in egrep
egrep: fancier regular expressions

\[ r^+ \text{ one or more occurrences of } r \]
\[ r^? \text{ zero or one occurrences of } r \]
\[ r_1 | r_2 \text{ } r_1 \text{ or } r_2 \]
\[ (r) \text{ } r \text{ (grouping)} \]

grammar:
\[ r: c . ^ $ [ccc] [^ccc] \]
\[ r^* \text{ } r^+ \text{ } r^? \]
\[ r_1 \text{ } r_2 \]
\[ r_1 | r_2 \]
\[ (r) \]

precedence:
\[ * \text{ } + \text{ } ? \text{ higher than concatenation, which is higher than } l \]
The grep family

• grep
• egrep
  – fancier regular expressions, trades compile time and space for run time
• fgrep
  – parallel search for many fixed strings
• agrep
  – "approximate" grep: search with errors permitted
• relatives that use similar regular expressions
  – ed original Unix editor
  – sed stream editor
  – vi, emacs, sam, ... editors
  – lex, flex lexical analyzer generator
  – awk, perl, python, ... all scripting languages
  – Java, C# ... libraries in mainstream languages
• simpler variants
  – filename "wild cards" in Unix and other shells
  – "LIKE" operator in SQL, Visual Basic, etc.
Basic grep algorithm

while (get a line)
  if match(regexpr, line)
    print line

• (perhaps) compile regexpr into an internal representation suitable for efficient matching
• match() slides the line past the regexpr (or vice versa), looking for a match at each point
Match anywhere on a line

• look for match at each position of text in turn

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
Match starting at current position

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

• follow the easy case first: no metacharacters
• note that this is recursive
  – maximum depth: one level for each regexpr character that matches
Simple grep algorithm

- **best for short simple patterns**
  - e.g., `grep printf *.[ch]`
  - most use is like this
  - reflects use in text editor for a small machine

- **limitations**
  - tries the pattern at each possible starting point
    e.g., look for aaaaab in aaaa....aaaab
    potentially $O(mn)$ for pattern of length $m$
  - complicated patterns (\.*\.*\.*) require backup
    potentially exponential
  - can't do some xys, like alternation (OR)

- **this leads to extensions and new algorithms**
  - egrep complicated patterns, alternation
  - fgrep lots of simple patterns in parallel
  - boyer-moore long simple patterns
  - agrep approximate matches
Important ideas from regexprs & grep

• **tools**: let the machine do the work
  – good packaging matters

• **notation**: makes it easy to say what to do
  – may organize or define implementation

• hacking can make a program faster, sometimes, usually at the price of more complexity

• a better algorithm can make a program go a lot faster

• don't worry about performance if it doesn't matter (and it often doesn't)

• when it does,
  – use the right algorithm
  – use the compiler's optimization
  – code tune, as a last resort